ARMY RESEARCH LABORATORY

Creating, Searching, and Deleting KD Trees Using C++

by Robert J Yager

ARL-TN-0629 September 2014

NOTICES

Disclaimers

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TN-0629 **September 2014**

Creating, Searching, and Deleting KD Trees Using C++

by Robert J Yager
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2014	Final	November 2013–June 2014

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Creating, Searching, and Deleting KD Trees Using C++	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	**5d. PROJECT NUMBER**
Robert J Yager	AH80
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	**8. PERFORMING ORGANIZATION REPORT NUMBER**
US Army Research Laboratory ATTN: RDRL-WML-A Aberdeen Proving Ground, MD 21005-5066	ARL-TN-0629
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

K-dimensional (KD) trees use binary space-partitioning algorithms to organize and store data points in K-dimensional space. They are particularly useful for efficient nearest-neighbor search algorithms. This report presents a set of functions, written in C++, that is designed to be used to create, search, and delete KD trees. All of the functions are based on recursive algorithms. Tests for measuring function performance are included, as are examples for creating Voronoi diagrams.

15. SUBJECT TERMS

KD tree, Voronoi, Fermat, C++, code

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					Robert J Yager
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** (*Include area code*)
Unclassified	Unclassified	Unclassified	UU	28	410-278-6689

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

Acknowledgments

I would like to thank Tim Fargus of the US Army Armament Research, Development and Engineering Center's System Analysis Division. Tim provided technical and editorial recommendations that improved the quality of this report.

1. Introduction

K-dimensional (KD) trees use binary space-partitioning algorithms to organize and store data points in K-dimensional space.[1] They are particularly useful for efficient nearest-neighbor search algorithms. This report presents a set of functions, written in C++, that is designed to be used to create, search, and delete KD trees. All of the functions are based on recursive algorithms. Tests for measuring function performance are included, as are examples for creating Voronoi diagrams.

The functions that are described in this report have been grouped into the yKDTree namespace, which is summarized at the end of this report. The yKDTree namespace relies exclusively on standard C++ operations. However, example code that is included in this report makes use of the yRandom namespace[2] for generating pseudorandom numbers and the yBmp namespace[3] for creating Voronoi diagrams.

2. Sorting Tables — the ColumnSort() Function

The ColumnSort() function uses a stable merge-sort algorithm to sort tabulated data into ascending order. The data must be stored in a 2-index array, where the indices are arbitrarily referred to here as rows (first index) and columns (second index).

The ColumnSort() function is included in the yKDTree namespace as a helper function for the NewTree() function, which is described in section 4. However, the ColumnSort() function can also be useful on its own when an efficient means of sorting tables by columns is desired.

2.1 ColumnSort() Code

```
template<class T>void ColumnSort(//<=SORT A TABLE BY COLUMNS (ASCENDING ORDER)
    T**a,T**b,//<-------------POINTERS TO STARTING & ENDING ROWS (FIRST INDEX)
    T**t,//<----------------------------------TEMPORARY STORAGE (SIZE >= b-a)
    unsigned c){//<-------------------------THE COLUMN (2ND INDEX) TO SORT ON
  if(b-a<2)return;
  T**m=a+(b-a)/2;/*->*/ColumnSort(a,m,t,c),ColumnSort(m,b,t,c);
  for(T**u=m,**l=a,**j=t;j<b-a+t;)*j++=u<b&&(l>=m||(*l)[c]>(*u)[c])?*u++:*l++;
  while(a<b)*a++=*t++;
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

2.2 ColumnSort() Parameters

a — **a** points to the first row that will be included in the sort.

b — **b** points to one past the last row that will be included in the sort.

t — **t** points to temporary storage for the ColumnSort() function. **t** must point to a array that is capable of storing at least **b-a** elements.

c — **c** specifies the column on which the sort will be based.

2.3 ColumnSort() Example

The following example uses the ColumnSort() function to sort a 2-column table, first by the second column, then by the first.

```
#include <cstdio>//..................................................printf()
#include "y_kd_tree.h"//..............................................yKDTree
int main(){//<====================A SIMPLE EXAMPLE FOR THE ColumnSort() FUNCTION
  int *T[9],A[9][2]={0,5,0,7,1,9,0,8,1,6,0,7,1,4,0,0,0,3},*B[9];
  for(int i=0;i<9;++i)B[i]=A[i];
  yKDTree::ColumnSort(B,B+9,T,1),yKDTree::ColumnSort(B,B+9,T,0);
  printf(" UNSORTED |  SORTED\n ---------|--------\n");
  for(int i=0;i<9;++i)
    printf("   %d , %d  |  %d , %d\n",*A[i],A[i][1],*B[i],B[i][1]);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

OUTPUT:

```
UNSORTED |  SORTED
 ---------|--------
   0 , 5  |  0 , 0
   0 , 7  |  0 , 3
   1 , 9  |  0 , 5
   0 , 8  |  0 , 7
   1 , 6  |  0 , 7
   0 , 7  |  0 , 8
   1 , 4  |  1 , 4
   0 , 0  |  1 , 6
   0 , 3  |  1 , 9
```

2.4 ColumnSort() Performance

The following code measures the performance of the ColumnSort() function and compares it with the performance of the stable_sort() function. The yRandom namespace is used to generate pseudorandom numbers for the test. Time measurements represent the total time required to perform 10^6 sorts for tables with 2^n rows, where n varies from 1 to 14.

The output was generated by compiling the code using Microsoft's Visual Studio C++ 2010 Express compiler, with the output set to "release" mode. For this scenario, the ColumnSort() function outperforms the built-in stable_sort() function (Fig. 1).

```
#include <algorithm>//............................................stable_sort()
#include <cstdio>//.....................................................printf()
#include <ctime>//.........................................clock(),CLOCKS_PER_SEC
#include "y_kd_tree.h"//...................................................yKDTree
#include "y_random.h"//....................................................yRandom
inline bool Compare(//<======HELPER FUNCTION FOR THE std::stable_sort() FUNCTION
    double*a,double*b){//<--------------------------POINTERS TO COMPARISON ROWS
  return a[0]<b[0];//..........................note column number is fixed at 0
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<==============MEASURE THE PERFORMANCE OF THE ColumnSort() FUNCTION
  const int N=1<<14,M=1000000;//....max # of rows, number of iterations per test
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  double s,t,*T[N],*A[N];/*<-*/for(int i=0;i<N;++i)A[i]=new double[1];
  printf("          |  std::stable_sort()  |     ColumnSort()\n"
    "   row    |----------------------|----------------------\n"
    "  count   |    time    | Z[m/2][0] |    time    | Z[m/2][0]\n"
    "          |    (s)     |    avg.   |    (s)     |    avg.\n"
    " ---------|----------|-----------|----------|-----------\n");//table header
  for(int m=2;m<=N;m*=2){
    s=0,t=clock(),yRandom::Initialize(I,1);//..........begin stable_sort() test
    for(int k=0;k<M;++k){
      for(int i=0;i<m;++i)A[i][0]=yRandom::RandU(I,0,1);
      std::stable_sort(A,A+m,Compare),s+=A[m/2][0];}
    printf("%7d   |%9.3f |%10.7f |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock(),yRandom::Initialize(I,1);//...........begin ColumnSort() test
    for(int k=0;k<M;++k){
      for(int i=0;i<m;++i)A[i][0]=yRandom::RandU(I,0,1);
      yKDTree::ColumnSort(A,A+m,T,0),s+=A[m/2][0];}
    printf("%9.3f |%10.7f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
  for(int i=0;i<N;++i)delete[]A[i];
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

OUTPUT:

```
           | std::stable_sort()    |     ColumnSort()
   row     |----------------------|----------------------
   count   |   time    | Z[m/2][0] |   time    | Z[m/2][0]
           |   (s)     |   avg.    |   (s)     |   avg.
---------|----------|-----------|----------|-----------
       2   |    0.046 | 0.6667224 |    0.032 | 0.6667224
       4   |    0.078 | 0.6002211 |    0.093 | 0.6002211
       8   |    0.219 | 0.5556498 |    0.249 | 0.5556498
      16   |    0.546 | 0.5293284 |    0.577 | 0.5293284
      32   |    1.316 | 0.5150998 |    1.263 | 0.5150998
      64   |    3.401 | 0.5077139 |    2.886 | 0.5077139
     128   |    7.675 | 0.5038248 |    6.646 | 0.5038248
     256   |   17.456 | 0.5019407 |   14.682 | 0.5019407
     512   |   39.059 | 0.5009805 |   31.961 | 0.5009805
    1024   |   83.569 | 0.5004688 |   68.828 | 0.5004688
    2048   |  186.202 | 0.5002375 |  150.859 | 0.5002375
    4096   |  409.029 | 0.5001205 |  324.842 | 0.5001205
    8192   |  890.498 | 0.5000599 |  697.758 | 0.5000599
   16384   | 2061.582 | 0.5000335 | 1537.217 | 0.5000335
```

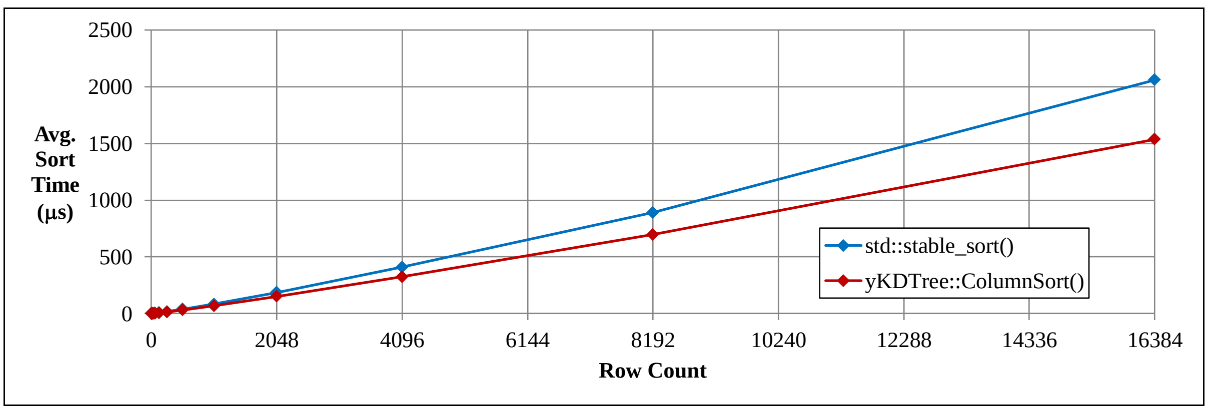


Fig. 1 ColumnSort() performance compared with stable_sort() performance

3. KD-Tree Nodes — the NODE Struct

NODE structs can be used to store the nodes that make up KD trees. Typically, NODE structs are created using the NewTree() function (see section 4) and deleted using the DeleteTree() function (see section 5).

3.1 NODE Code

```
template<class T>struct NODE{//<================================A KD-TREE NODE
  T*r;//<------------------------------------------POINTER TO A ROW IN A TABLE
  NODE*a,*b;//<-------------------------------------------POINTERS TO SUBNODES
  int k;//<---------------------------------------------------------NODE LAYER
};//~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

3.2 NODE Parameters

r **r** points to a row in a table.

a **a** points to a subnode.

b **b** points to a subnode.

k **k** is used to identify a node's layer.

4. Creating KD Trees — the NewTree() Function

If a 2-index array is used to store sortable tabulated data in the format that is shown in Fig. 2, then the NewTree() function can be used to create a KD tree for the tabulated data. In Fig. 2, values for both indepenent and dependent variables are stored in array A. Independent variables are stored in columns with subscripted-X headers, while dependent variables are stored in columns with subscripted-Y headers. Each row represents a single data point.

	X_0	X_1	$\cdots$	X_k	$\cdots$	X_{K-1}	Y_0	Y_1	$\cdots$
0	$A_{0,0}$	$A_{0,1}$	$\cdots$	$A_{0,k}$	$\cdots$	$A_{0,K-1}$	$A_{0,K}$	$A_{0,K+1}$	$\cdots$
1	$A_{1,0}$	$A_{1,1}$	$\cdots$	$A_{1,k}$	$\cdots$	$A_{1,K-1}$	$A_{1,K}$	$A_{1,K+1}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	
i	$A_{i,0}$	$A_{i,1}$	$\cdots$	$A_{i,k}$	$\cdots$	$A_{i,K-1}$	$A_{i,K}$	$A_{i,K+1}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	
$m-1$	$A_{m-1,0}$	$A_{m-1,1}$	$\cdots$	$A_{m-1,k}$	$\cdots$	$A_{m-1,K-1}$	$A_{m-1,K}$	$A_{m-1,K+1}$	$\cdots$

Fig. 2 Tabulated data stored in a 2-index array

Note that the NewTree() function uses the "new" command to allocate memory for nodes. To avoid memory leaks, the DeleteTree() function (see section 5) should be used to deallocate memory when a KD tree is no longer needed.

4.1 NewTree() Code

```
template<class T>NODE<T>*NewTree(//<========================CREATE A KD TREE
    T**a,T**b,//<------------------------POINTERS TO STARTING AND ENDING ROWS
    int K,//<--------------------------------NUMBER OF INDEPENDENT DIMENSIONS
    int k=0){//<----------------------NODE LAYER (SET BY RECURSIVE ALGORITHM)
  T**t=new T*[b-a];/*->*/ColumnSort(a,b,t,k);/*&*/delete[]t;
  T**m=a+(b-a-1)/2;
  NODE<T>*N=new NODE<T>;/*<-*/N->r=*m,N->a=a-m?NewTree(a,m,K,(k+1)%K):0,
    N->b=b-m-1?NewTree(m+1,b,K,(k+1)%K):0,N->k=k;
  return N;
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

4.2 NewTree() Parameters

a — **a** points to the first row of the table that will be included in the new KD tree.

b — **b** points to one past the last row of the table that will be included in the new KD tree.

K — **K** specifies the number of independent dimensions that are included in the table that is specified by **a** and **b**.

k — **k** specifies the layer of the current node that is being created by the NewTree() function. **k** is set automatically, either by the default value, or by the NewTree() function when it calls itself recursively.

4.3 NewTree() Return Value

The NewTree() function returns a pointer to the root node of a newly created KD tree.

4.4 NewTree() Example — Creating a Simple KD Tree

The following example code uses the NewTree() function to create a simple KD tree, with $K = 2$ independent dimensions, then prints the nodes. Nodes represented by (X,X) are empty.

```
#include <cstdio>//....................................................printf()
#include "y_kd_tree.h"//................................................yKDTree
inline void PrintNode(//<======================PRINTS THE COORDINATES OF A NODE
    yKDTree::NODE<int>*N){//<---------------------------------------------A NODE
  N?printf("(%d,%d)   ",N->r[0],N->r[1]):printf("(X,X)   ");
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<=====================A SIMPLE EXAMPLE FOR THE NewKDTree() FUNCTION
  int*A[5],B[5][3]={4,1,0 , 4,3,1 , 6,2,2 , 2,4,3 , 8,4,4};
  printf("TABULATED DATA:\n");
  for(int i=0;i<5;++i)printf("%13d,%d,%d\n",B[i][0],B[i][1],B[i][2]);
  printf("\n\nKD TREE:\n");
  for(int i=0;i<5;++i)A[i]=B[i];
  yKDTree::NODE<int>*N=yKDTree::NewTree(A,A+5,2);
  printf("            "),PrintNode(N);//........................root node (k=0)
  printf("\n            /   \\\n    ");
  PrintNode(N->a),printf("        ");PrintNode(N->b);//....2nd-level nodes (k=1)
  printf("\n    /   \\          /   \\\n");
  PrintNode(N->a->a),PrintNode(N->a->b),PrintNode(N->b->a),PrintNode(N->b->b);
  printf("\n\n"),yKDTree::DeleteTree(N);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

OUTPUT:

```
TABULATED DATA:
            4,1,0
            4,3,1
            6,2,2
            2,4,3
            8,4,4


KD TREE:
            (4,3)
            /   \
    (4,1)           (6,2)
    /   \           /   \
(X,X)   (2,4)   (X,X)   (8,4)
```

---

5. Deleting KD Trees — the DeleteTree() Function

The DeleteTree() function can be used to delete a KD tree that has been created using the NewTree() function.

5.1 DeleteTree() Code

```
template<class T>void DeleteTree(//<========================DELETE A KD TREE
    NODE<T>*N){//<------------------------------------THE ROOT NODE OF A KD TREE
  if(!N)return;/*else*/DeleteTree(N->a),DeleteTree(N->b),delete N;
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

5.2 DeleteTree() Parameters

N **N** points to the root node of a KD tree.

6. Searching KD Trees — the NNSearch() Function

The NNSearch() function can be used to search a KD tree for A_i, a row in a table that specifies a location that minimizes the distance to X, a user specified location where

$$X = \{X_0, X_1, \cdots, X_k, \cdots, X_{K-1}\} . \tag{1}$$

Thus, A_i is defined to be a row for which S in Eq. 2 is minimized.

$$S = \sum_{k=0}^{K-1} \left(X_k - A_{i,k}\right)^2 . \tag{2}$$

Note that A_i may or may not be unique.

6.1 NNSearch() Code

```
template<class T,class U>void NNSearch(//<=============NEAREST-NEIGHBOR SEARCH
    const NODE<T>*N,//<-----------------------------THE ROOT NODE OF A KD TREE
    const T*X,//<--------------------------------POINTER TO SEARCH COORDINATES
    T*&r,//<----------------------POINTER TO NEAREST-NEIGHBOR ROW (CALCULATED)
    U&S,//<-------SQUARED DISTANCE BETWEEN COORDINATES AT X AND r (CALCULATED)
    int K){//<-------------------------------NUMBER OF INDEPENDENT DIMENSIONS
  NODE<T>*a,*b;/*<-*/X[N->k]<N->r[N->k]?a=N->a,b=N->b:(a=N->b,b=N->a);
  if(a)NNSearch(a,X,r,S,K);
  U s=0;/*<-*/for(int k=0;k<K;++k)s+=(X[k]-N->r[k])*(X[k]-N->r[k]);
  if(s<S)r=N->r,S=s;
  if((s=X[N->k]-N->r[N->k])*s<S&&b)NNSearch(b,X,r,S,K);
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

6.2 NNSearch() Parameters

N **N** points to the root node of a KD tree.

X **X** points to X, the coordinates of the location for the search. The coordinates that are pointed to by **X** must consist of **K** values that correspond to the **K** independent variables that are associated with the KD tree specified by **N**.

r **r** points to A_i, a row that specifies the coordinates of a location for which the distance to the location specified by **X** is minimized. **r** is calculated by the NNSearch() function.

S **S** specifies S, the squared distance between the locations specified by **X** and **r**. Although **S** is calculated by the NNSearch() function, **S** should be initialized to some value that is larger than the expected final value of **S** (typically some very large value).

K **K** specifies K, the number of independent dimensions that are included in the KD tree.

6.3 NNSearch() Example — Creating a Simple Voronoi Diagram

The following example code uses functions from the yBmp namespace, along with the NNSearch() function to create the Voronoi diagram that is presented in Fig. 3. The black dots in Fig. 3 show the locations of the points that were used to create the Voronoi diagram. The colored sections represent sets of points that have a common nearest neighbor among the points that make up the KD tree (i.e., the black dots). For this particular Voronoi diagram, the colors themselves represent the row number of the table that was used to create the KD tree (see Fig. 4).

```
#include <cmath>//.......................................................fabs()
#include "y_bmp.h"//....................................yBmp,<cstring>{memcpy()}
#include "y_kd_tree.h"//...................................................yKDTree
inline void Rainbow(//<=========================================RAINBOW COLOR MAP
    unsigned char C[3],//<----------------------------OUTPUT COLOR (CALCULATED)
    double x,//<----------------------VALUE FOR WHICH A COLOR WILL BE CALCULATED
    double min,double max){//<-----------------MINIMUM AND MAXIMUM SCALED VALUES
  if(x<min){C[0]=C[1]=C[2]=0;/*&*/return;}//.......set too small values to black
  if(x>max){C[0]=C[1]=C[2]=255;/*&*/return;}//.....set too large values to white
  x=(1-(x-min)/(max-min))*8;//......................remap x to a range of 8 to 0
  C[0]=int((3<x&&x<5||x>7     ?-fabs(x/2-3)+1.5:5<=x&&x<=7?1:0)*255);//.....blue
  C[1]=int((1<x&&x<3||5<x&&x<7?-fabs(x/2-2)+1.5:3<=x&&x<=5?1:0)*255);//....green
  C[2]=int((     x<1||3<x&&x<5?-fabs(x/2-1)+1.5:1<=x&&x<=3?1:0)*255);//......red
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<========================CREATE A VORONOI DIAGRAM USING NNSearch()
  double*A[5],B[5][3]={4,1,0 , 4,3,1 , 6,2,2 , 2,4,3 , 8,4,4};
  for(int i=0;i<5;++i)A[i]=B[i];
  yKDTree::NODE<double>*N=yKDTree::NewTree(A,A+5,2);
  int n=1000;//................................image size will be 2n x n pixels
  unsigned char*I=yBmp::NewImage(2*n,n,255),BLACK[3]={0};
  for(int p=0;p<2*n;++p)for(int q=0;q<n;++q){
    double X[2]={p*10./(2*n-1),q*5./(n-1)};
    double S,*r;/*<-*/yKDTree::NNSearch(N,X,r,S=1E9,2);
    if(S<.002)memcpy(yBmp::GetPixel(I,p,q),BLACK,3);
    else Rainbow(yBmp::GetPixel(I,p,q),r[2],-1.5,5.5);}
  yBmp::WriteBmpFile("voronoi.bmp",I);
  yKDTree::DeleteTree(N),delete[]I;
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

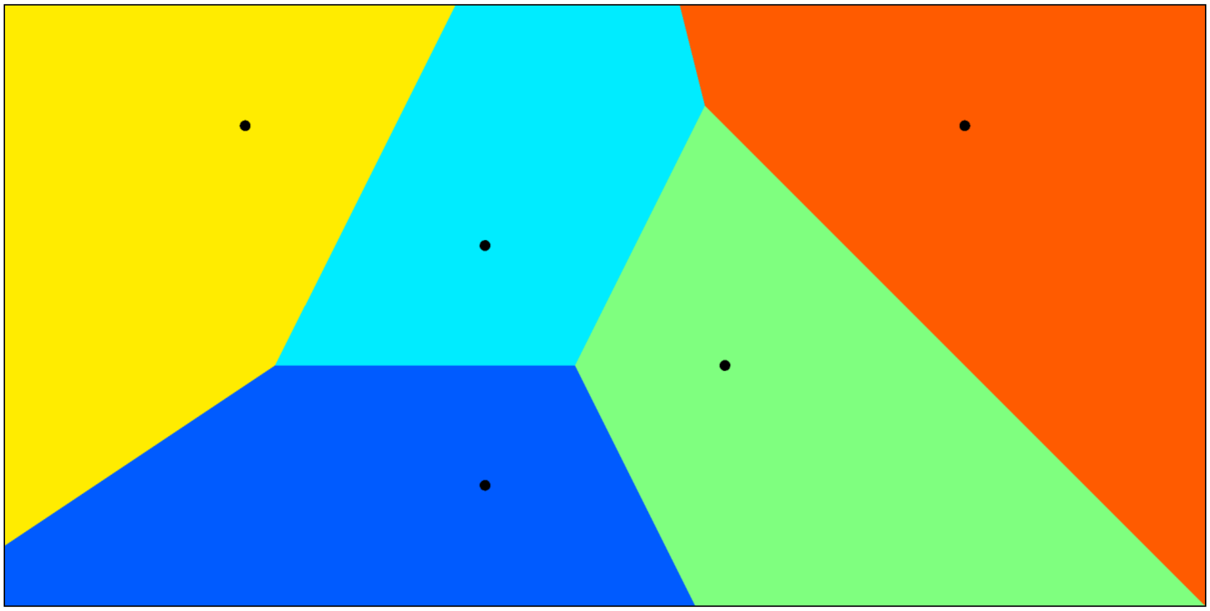


Fig. 3   A simple Voronoi diagram

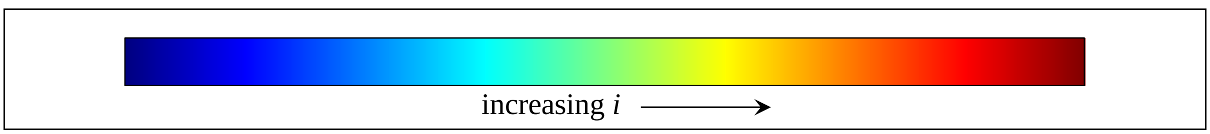


Fig. 4   Scale for Voronoi diagrams

6.4 NNSearch() Performance

The following code measures the performance of the NNSearch() function and compares it with the performance of the NNSearchExhaustive() function, which is defined in the example. The NNSearchExhaustive() function takes a brute-force approach to determining A_i and S .

The yRandom namespace is used to generate pseudorandom numbers for the test. Time measurements represent the total time required to perform 10^7 searches on tables with 2^n rows, where n varies from 1 to 14.

When $K = 2$, the test shows that for tables with very few rows (somewhere around 32 or fewer) the brute-force method outperforms the KD-tree method. Figure 5 shows that as the value of K increases, the minimum number of rows required for the KD-tree method to be advantageous increases as well.

```
#include <cstdio>//..................................................printf()
#include <ctime>//..........................................clock(),CLOCKS_PER_SEC
#include "y_kd_tree.h"//................................................yKDTree
#include "y_random.h"//.................................................yRandom
template<class T>T NNSearchExhaustive(//<=====EXHAUSTIVE NEAREST-NEIGHBOR SEARCH
    T**a,T**b,//<---------------------------POINTERS TO STARTING AND ENDING ROWS
    T*X,//<-----------------------------------------POINTER TO SEARCH COORDINATES
    T*&r,//<------------------------POINTER TO NEAREST-NEIGHBOR ROW (CALCULATED)
    int K){//<----------------------------------NUMBER OF INDEPENDENT DIMENSIONS
  double S=0;/*<-*/for(int i=0;i<K;++i)S+=(a[0][i]-X[i])*(a[0][i]-X[i]);
  for(r=*a;a<b;++a){
    double s=0;/*<-*/for(int i=0;i<K;++i)s+=((*a)[i]-X[i])*((*a)[i]-X[i]);
    if(s<S)r=*a,S=s;}
  return S;
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<================MEASURE THE PERFORMANCE OF THE NNSearch() FUNCTION
  const int N=1<<14,M=10000000,K=2;
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  double*A[N],B[N][K];/*<-*/for(int i=0;i<N;++i)A[i]=B[i];
  for(int i=0;i<N;++i)for(int k=0;k<K;++k)A[i][k]=yRandom::RandU(I,0,1);
  printf("          | NNSearchExhaustive()|     NN2DInterp()\n"
    "   row    |---------------------|---------------------\n"
    "  count   |    time   |     x     |    time   |     x\n"
    "          |     (s)   |    avg.   |     (s)   |    avg.\n"
    " ---------|----------|----------|----------|----------\n");//..table header
  for(int m=2;m<=N;m*=2){
    double s=0,t=clock();/*&*/yRandom::Initialize(I,1);//...NNSearchExhaustive()
    for(int k=0;k<M;++k){
      double X[K];/*<-*/for(int i=0;i<K;++i)X[i]=yRandom::RandU(I,0,1);
      double*r;/*<-*/NNSearchExhaustive(A,A+m,X,r,K);
      s+=*r;}
    printf("%7d    |%8.3f   |%9.6f |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock(),yRandom::Initialize(I,1);//..............begin NNSearch() test
    yKDTree::NODE<double>*R=yKDTree::NewTree(A,A+m,K);
    for(int k=0;k<M;++k){
      double X[K];/*<-*/for(int i=0;i<K;++i)X[i]=yRandom::RandU(I,0,1);
      double S=1E9;
      double*r;/*<-*/yKDTree::NNSearch(R,X,r,S,K);
      s+=*r;}
    yKDTree::DeleteTree(R);
    printf("%8.3f  |%9.6f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

OUTPUT:

```
          | NNSearchExhaustive()|      NN2DInterp()
  row     |---------------------|---------------------
  count   |   time    |    x    |   time    |    x
          |    (s)    |   avg.  |    (s)    |   avg.
----------|-----------|---------|-----------|----------
        2 |   0.320   | 0.577847 |   0.390   | 0.577847
        4 |   0.380   | 0.372263 |   0.646   | 0.372263
        8 |   0.552   | 0.354816 |   0.801   | 0.354816
       16 |   0.710   | 0.481116 |   0.920   | 0.481116
       32 |   1.050   | 0.491685 |   1.200   | 0.491685
       64 |   1.760   | 0.499086 |   1.506   | 0.499086
      128 |   3.104   | 0.499204 |   1.760   | 0.499204
      256 |   5.630   | 0.498995 |   2.060   | 0.498995
      512 |  11.131   | 0.499451 |   2.360   | 0.499451
     1024 |  21.392   | 0.499869 |   2.610   | 0.499869
     2048 |  43.283   | 0.499950 |   2.980   | 0.499950
     4096 |  88.706   | 0.499952 |   3.091   | 0.499952
     8192 | 177.382   | 0.499965 |   3.450   | 0.499965
    16384 | 382.268   | 0.499971 |   3.880   | 0.499971
```

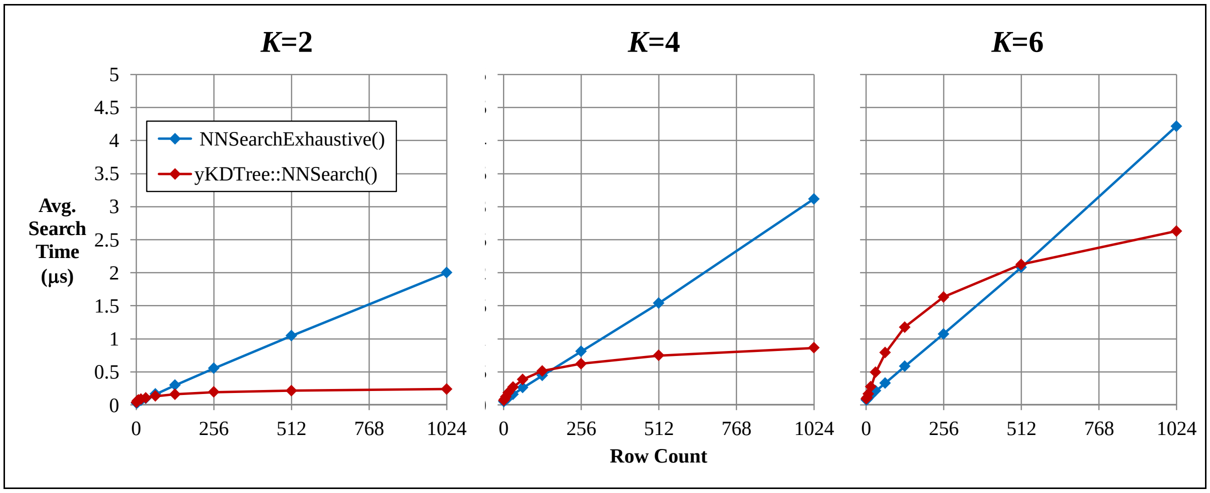


Fig. 5 Brute-force and KD-tree methods compared for K = 2, 4, and 6

7. Example — Fermat's Spiral

According to Vogel[4] Eqs. 3 and 4, which define a set of points in polar coordinates that all lie on Fermat's spiral, can be used to model the patterns of seeds in sunflowers.

$$r_i = R\sqrt{\frac{i}{m-1}}, \tag{3}$$

$$\theta_i = \left(3-\sqrt{5}\right)i\pi, \tag{4}$$

where R is the radius of a circle that contains all of the points, m is the total number of points, and $0 \le i < m$.

The following code uses functions from the yBmp namespace, as well as the Rainbow() function from section 6.3, to create three Voronoi diagrams (Fig. 6) that are based on Eqs. 3 and 4 (one with m =100, one with m =200, and one with m =600). For each, R has been chosen to be 1/2 the width of the image.

```
#include <cmath>//.......................................................sqrt()
#include "y_bmp.h"//...................................yBmp,<cstring>{memcpy()}
#include "y_kd_tree.h"//............................................yKDTree
int main(){//<==================CREATE VORONOI DIAGRAMS BASED ON FERMAT'S SPIRAL
  for(int m=100,j=0,n=1000;j<3;++j,m*=j+1){//....image size will be n x n pixels
    double**A=new double*[m];/*<-*/for(int i=0;i<m;++i)A[i]=new double[3];
    for(int i=0;i<m;++i){
      double R=.5*sqrt(i/(m-1.)),theta=(3-sqrt(5.))*i*3.141592653589793;
      double x=R*cos(theta),y=R*sin(theta);
      A[i][0]=x+.5,A[i][1]=y+.5,A[i][2]=i;}
    yKDTree::NODE<double>*N=yKDTree::NewTree(A,A+m,2);
    unsigned char*B=yBmp::NewImage(n,n,255),BLACK[3]={0};
    for(int p=0;p<n;++p)for(int q=0;q<n;++q){
      double X[2]={p/(n-1.),q/(n-1.)};
      double S,*r;/*<-*/yKDTree::NNSearch(N,X,r,S=1E9,2);
      if(S<.00002)memcpy(yBmp::GetPixel(B,p,q),BLACK,3);
      else Rainbow(yBmp::GetPixel(B,p,q),r[2],0,m);}
    yBmp::WriteBmpFile(j==0?"spiral1.bmp":j==1?"spiral2.bmp":"spiral3.bmp",B);
    for(int i=0;i<m;++i)delete[]A[i];/*&*/yKDTree::DeleteTree(N),delete[]B;}
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

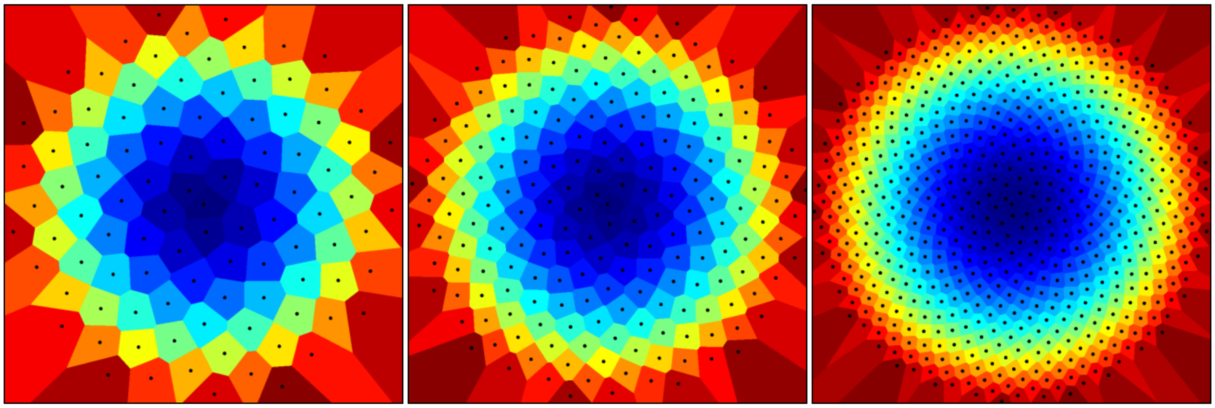


Fig. 6 Voronoi diagrams of Fermat's Spiral (m = 100 left, m = 200 center, m = 600 right)

8. Example — Comparing Average Distances to Sources

Suppose that the effectiveness of some physical phenomenon of interest is reduced as the distance from a source increases (such as signal strength from radio antennas). If $D(x, y)$ is defined to be the distance to the nearest source, then Eq. 5 can be used to calculate the average distance to the nearest source ($\overline{D}$) for some area of interest:

$$\overline{D} \equiv \frac{\int\int D(x, y)dxdy}{A}, \tag{5}$$

where A is the total area over which $D(x, y)$ is integrated. By comparing $\overline{D}$ values for different sets of points, the relative coverage effectiveness between sets can be evaluated.

The following example code uses functions from the yBmp namespace, as well as the Rainbow() function from section 6.3, to create 3 Voronoi diagrams (Fig. 7). The first 2 are based on tables containing randomly selected points, while the third is based on a set of points that was purposely chosen to result in a small value for $\overline{D}$.

```cpp
#include "y_kd_tree.h"//.............................................yKDTree
#include "y_bmp.h"//...................................yBmp,<cstring>{memcpy()}
#include "y_random.h"//..............................................yRandom
int main(){//<==========================================COMPARE AREAL COVERAGES
  const int m=9,n=1000;//.......................image size will be n x n pixels
  double*A[m];/*<-*/for(int i=0;i<m;++i)A[i]=new double[3];
  double d=1./6;
  double O[m][2]={3*d,3*d,5*d,3*d,5*d,5*d,3*d,5*d,d,5*d,d,3*d,d,d,3*d,d,5*d,d};
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  unsigned char*B=yBmp::NewImage(n,n,255),BLACK[3]={0};
  for(int J=0;J<3;++J){
    for(int i=0;i<m;++i)
      A[i][0]=J==2?O[i][0]:yRandom::RandU(I,0,1),
      A[i][1]=J==2?O[i][1]:yRandom::RandU(I,0,1),
      A[i][2]=i;
    yKDTree::NODE<double>*N=yKDTree::NewTree(A,A+m,2);
    double s=0;
    for(int p=0;p<n;++p)for(int q=0;q<n;++q){
      double X[2]={p*1./(n-1),q*1./(n-1)};
      double S,*r;/*<-*/yKDTree::NNSearch(N,X,r,S=1E9,2);
      s+=sqrt(S);
      if(S<.00002)memcpy(yBmp::GetPixel(B,p,q),BLACK,3);
      else Rainbow(yBmp::GetPixel(B,p,q),r[2],-1,m);}
    yBmp::WriteBmpFile(!J?"coverage1.bmp":J==1?"coverage2.bmp":
      "coverage3.bmp",B);
    printf("CASE %d: D_bar=%8.5f\n",J+1,s/n/n);
    yKDTree::DeleteTree(N);}
  for(int i=0;i<m;++i)delete[]A[i];/*&*/delete[]B;
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

OUTPUT:

```
CASE 1: D_bar= 0.27243
CASE 2: D_bar= 0.17623
CASE 3: D_bar= 0.12766
```

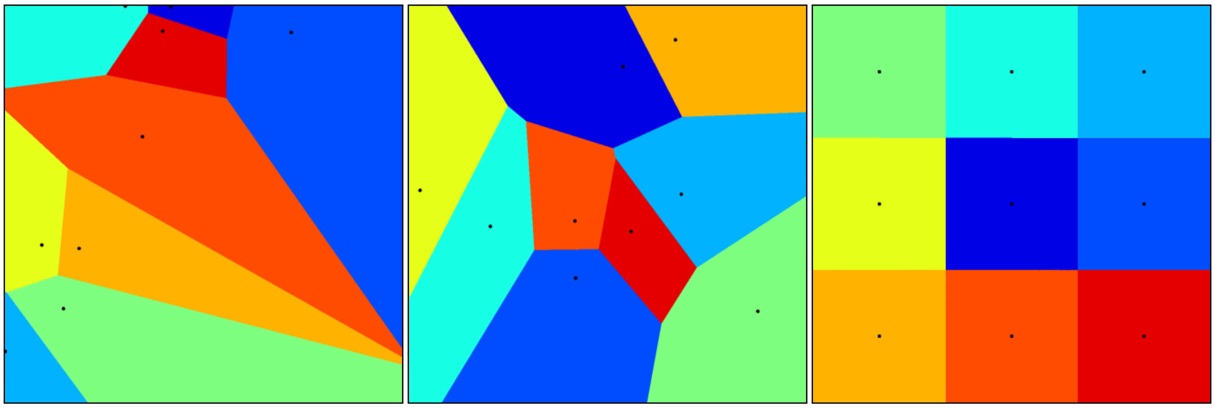


Fig. 7 Voronoi diagrams (case 1 left, case 2 center, case 3 right)

9. Example — Optimizing Areal Coverage

The following example code begins by recreating the “CASE 1” KD tree from the example presented in section 8. Next, a random-walk method is used to determine the optimal placement of an additional point. Figure 8 shows the original Voronoi diagram compared with the Voronoi diagram with the additional point.

The purpose of this example is to show a type of problem that might benefit from the use of KD trees. For this particular case, since the number of points in the table being searched is so small, it would likely have been slightly faster to use a brute-force method.

```
#include "y_kd_tree.h"//.................................................yKDTree
#include "y_bmp.h"//...................................yBmp,<cstring>{memcpy()}
#include "y_random.h"//..................................................yRandom
int main(){//<=========================================OPTIMIZE AREAL COVERAGE
  double*T[10],*A[10];/*<-*/for(int i=0;i<10;++i)A[i]=new double[3];
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  int n=1000;//.................................image size will be n x n pixels
  unsigned char*B=yBmp::NewImage(n,n,255),BLACK[3]={0};
  for(int i=0;i<10;++i)
    A[i][0]=yRandom::RandU(I,0,1),A[i][1]=yRandom::RandU(I,0,1),A[i][2]=i;
  double s=1E99,x=0,y=0,st,xt,yt;
  for(int J=0;J<100;++J){
    for(int i=0;i<10;++i)T[i]=A[i];
    st=0,xt=*T[9]=x+yRandom::RandN(I,0,.1),yt=T[9][1]=y+yRandom::RandN(I,0,.1);
    if(xt<0||xt>1||yt<0||yt>1)continue;
    yKDTree::NODE<double>*N=yKDTree::NewTree(T,T+10,2);
    for(int p=0;p<n;++p)for(int q=0;q<n;++q){
      double X[2]={p*1./(n-1),q*1./(n-1)},D,*r;
      yKDTree::NNSearch(N,X,r,D=1E9,2),st+=sqrt(D);}
    if(st<s)printf("J=%2d , D_bar=%f , x=%9.6f , y=%9.6f\n",J,
      (s=st)/n/n,x=xt,y=yt);
    yKDTree::DeleteTree(N);}
  A[9][0]=x,A[9][1]=y;
  yKDTree::NODE<double>*N=yKDTree::NewTree(A,A+10,2);
  for(int p=0;p<n;++p)for(int q=0;q<n;++q){
    double X[2]={p*1./(n-1),q*1./(n-1)};
    double D,*r;/*<-*/yKDTree::NNSearch(N,X,r,D=1E9,2);
    if(D<.00002)memcpy(yBmp::GetPixel(B,p,q),BLACK,3);
    else Rainbow(yBmp::GetPixel(B,p,q),r[2],-1,9);}
  yBmp::WriteBmpFile("optimized_coverage.bmp",B);
  yKDTree::DeleteTree(N),delete[]B;
  for(int i=0;i<10;++i)delete[]A[i];
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```

OUTPUT:

```
J= 4 , D_bar=0.262436 , x= 0.183841 , y= 0.031927
J= 7 , D_bar=0.256637 , x= 0.217818 , y= 0.054442
J= 8 , D_bar=0.241756 , x= 0.324290 , y= 0.017688
J=12 , D_bar=0.239188 , x= 0.303139 , y= 0.112939
J=13 , D_bar=0.238673 , x= 0.321162 , y= 0.061664
J=14 , D_bar=0.209072 , x= 0.597562 , y= 0.007720
J=18 , D_bar=0.189872 , x= 0.633593 , y= 0.136296
J=20 , D_bar=0.187058 , x= 0.876571 , y= 0.212578
J=23 , D_bar=0.183453 , x= 0.797085 , y= 0.401234
J=26 , D_bar=0.179227 , x= 0.711760 , y= 0.272756
J=28 , D_bar=0.179046 , x= 0.769481 , y= 0.303093
J=62 , D_bar=0.178932 , x= 0.752994 , y= 0.268364
```

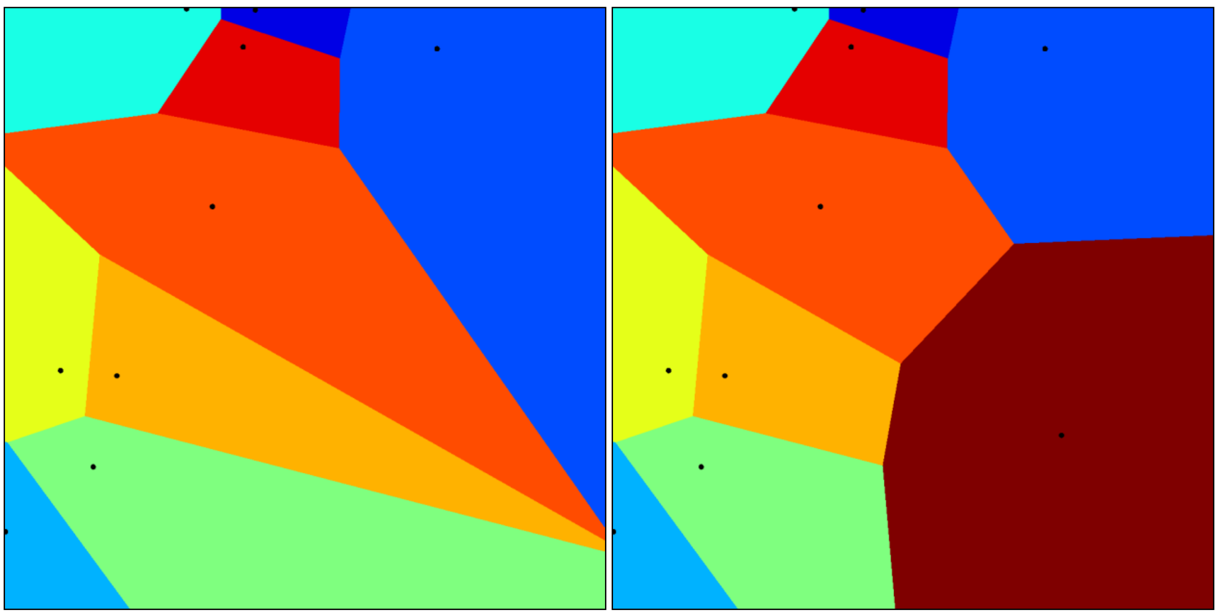


Fig. 8 Voronoi diagrams of a table without and with a point added that minimizes $\overline{D}$

10. Code Summary

A summary sheet is provided at the end of this report. It presents the yKDTree namespace, which contains the ColumnSort(), NewTree(), DeleteTree(), and NNSearch() functions and the NODE struct.

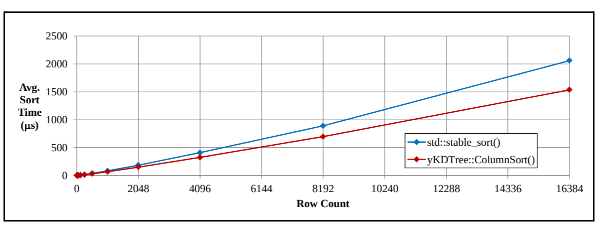
yKDTree Summary

y_kd_tree.h

```
#ifndef Y_KD_TREE_GUARD//                          See Yager, R.J. "Working with KD-Trees
#define Y_KD_TREE_GUARD//                                        Using C++" (ARL-TN-XXX)
namespace yKDTree{//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  template<class T>void ColumnSort(//<=SORT A TABLE BY COLUMNS (ASCENDING ORDER)
      T**a,T**b,//<-------------POINTERS TO STARTING & ENDING ROWS (FIRST INDEX)
      T**t,//<---------------------------------TEMPORARY STORAGE (SIZE >= b-a)
      unsigned c){//<--------------------------THE COLUMN (2ND INDEX) TO SORT ON
    if(b-a<2)return;
    T**m=a+(b-a)/2;/*->*/ColumnSort(a,m,t,c),ColumnSort(m,b,t,c);
    for(T**u=m,**l=a,**j=t;j<b-a+t;)*j++=u<b&&(l>=m||(*l)[c]>(*u)[c])?*u++:*l++;
    while(a<b)*a++=*t++;
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
  template<class T>struct NODE{//<================================A KD-TREE NODE
    T*r;//<-----------------------------------------------POINTER TO A ROW IN A TABLE
    NODE*a,*b;//<-------------------------------------------POINTERS TO SUBNODES
    int k;//<-----------------------------------------------------------NODE LAYER
  };//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
  template<class T>NODE<T>*NewTree(//<=========================CREATE A KD TREE
      T**a,T**b,//<-------------------------POINTERS TO STARTING AND ENDING ROWS
      int K,//<---------------------------------NUMBER OF INDEPENDENT DIMENSIONS
      int k=0){//<----------------------NODE LAYER (SET BY RECURSIVE ALGORITHM)
    T**t=new T*[b-a];/*->*/ColumnSort(a,b,t,k);/*&*/delete[]t;
    T**m=a+(b-a-1)/2;
    NODE<T>*N=new NODE<T>;/*<-*/N->r=*m,N->a=a-m?NewTree(a,m,K,(k+1)%K):0,
      N->b=b-m-1?NewTree(m+1,b,K,(k+1)%K):0,N->k=k;
    return N;
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
  template<class T>void DeleteTree(//<==========================DELETE A KD TREE
      NODE<T>*N){//<-----------------------------------THE ROOT NODE OF A KD TREE
    if(!N)return;/*else*/DeleteTree(N->a),DeleteTree(N->b),delete N;
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
  template<class T,class U>void NNSearch(//<=============NEAREST-NEIGHBOR SEARCH
      const NODE<T>*N,//<------------------------------THE ROOT NODE OF A KD TREE
      const T*X,//<-------------------------------POINTER TO SEARCH COORDINATES
      T*&r,//<---------------------POINTER TO NEAREST-NEIGHBOR ROW (CALCULATED)
      U&S,//<-------SQUARED DISTANCE BETWEEN COORDINATES AT X AND r (CALCULATED)
      int K){//<--------------------------------NUMBER OF INDEPENDENT DIMENSIONS
    NODE<T>*a,*b;/*<-*/X[N->k]<N->r[N->k]?a=N->a,b=N->b:(a=N->b,b=N->a);
    if(a)NNSearch(a,X,r,S,K);
    U s=0;/*<-*/for(int k=0;k<K;++k)s+=(X[k]-N->r[k])*(X[k]-N->r[k]);
    if(s<S)r=N->r,S=s;
    if((s=X[N->k]-N->r[N->k])*s<S&&b)NNSearch(b,X,r,S,K);
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
}//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#endif
```

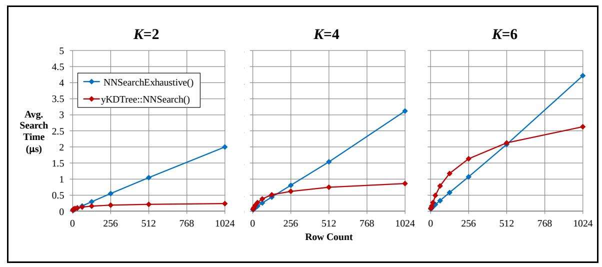
ColumnSort() Performance

```
#include <algorithm>//..........................................stable_sort()
#include <cstdio>//.....................................................printf()
#include <ctime>//......................................clock(),CLOCKS_PER_SEC
#include "y_kd_tree.h"//...............................................yKDTree
#include "y_random.h"//................................................yRandom
inline bool Compare(//<======HELPER FUNCTION FOR THE std::stable_sort() FUNCTION
    double*a,double*b){//<--------------------------POINTERS TO COMPARISON ROWS
  return a[0]<b[0];//..........................note column number is fixed at 0
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<==============MEASURE THE PERFORMANCE OF THE ColumnSort() FUNCTION
  const int N=1<<14,M=1000000;//....max # of rows, number of iterations per test
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  double s,t,*T[N],*A[N];/*<-*/for(int i=0;i<N;++i)A[i]=new double[1];
  printf("           |  std::stable_sort()  |     ColumnSort()\n"
    "   row     |---------------------|---------------------\n"
    "  count    |   time    | Z[m/2][0] |   time    | Z[m/2][0]\n"
    "           |    (s)    |   avg.    |    (s)    |   avg.\n"
    " ---------|----------|-----------|----------|-----------\n");//table header
  for(int m=2;m<=N;m*=2){
    s=0,t=clock(),yRandom::Initialize(I,1);//..........begin stable_sort() test
    for(int k=0;k<M;++k){
      for(int i=0;i<m;++i)A[i][0]=yRandom::RandU(I,0,1);
      std::stable_sort(A,A+m,Compare),s+=A[m/2][0];}
    printf("%7d   |%9.3f |%10.7f |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock(),yRandom::Initialize(I,1);//...........begin ColumnSort() test
    for(int k=0;k<M;++k){
      for(int i=0;i<m;++i)A[i][0]=yRandom::RandU(I,0,1);
      yKDTree::ColumnSort(A,A+m,T,0),s+=A[m/2][0];}
    printf("%9.3f |%10.7f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
  for(int i=0;i<N;++i)delete[]A[i];
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```



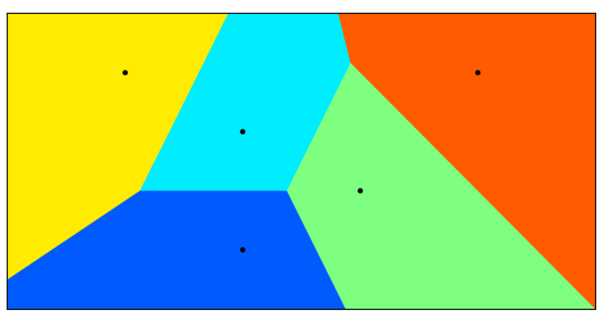
NNSearch() Performance

```
#include <cstdio>//.....................................................printf()
#include <ctime>//......................................clock(),CLOCKS_PER_SEC
#include "y_kd_tree.h"//...............................................yKDTree
#include "y_random.h"//................................................yRandom
template<class T>T NNSearchExhaustive(//<=====EXHAUSTIVE NEAREST-NEIGHBOR SEARCH
    T**a,T**b,//<--------------------------POINTERS TO STARTING AND ENDING ROWS
    T*X,//<-------------------------------------------POINTER TO SEARCH COORDINATES
    T*&r,//<-----------------------POINTER TO NEAREST-NEIGHBOR ROW (CALCULATED)
    int K){//<--------------------------------NUMBER OF INDEPENDENT DIMENSIONS
  double S=0;/*<-*/for(int i=0;i<K;++i)S+=(a[0][i]-X[i])*(a[0][i]-X[i]);
  for(r=*a;a<b;++a){
    double s=0;/*<-*/for(int i=0;i<K;++i)s+=((*a)[i]-X[i])*((*a)[i]-X[i]);
    if(s<S)r=*a,S=s;}
  return S;
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<================MEASURE THE PERFORMANCE OF THE NNSearch() FUNCTION
  const int N=1<<14,M=10000000,K=2;
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  double*A[N],B[N][K];/*<-*/for(int i=0;i<N;++i)A[i]=B[i];
  for(int i=0;i<N;++i)for(int k=0;k<K;++k)A[i][k]=yRandom::RandU(I,0,1);
  printf("           | NNSearchExhaustive()|     NN2DInterp()\n"
    "   row     |---------------------|---------------------\n"
    "  count    |   time    |    x     |   time    |    x\n"
    "           |    (s)    |   avg.   |    (s)    |   avg.\n"
    " ---------|----------|----------|----------|----------\n");//..table header
  for(int m=2;m<=N;m*=2){
    double s=0,t=clock();/*&*/yRandom::Initialize(I,1);//...NNSearchExhaustive()
    for(int k=0;k<M;++k){
      double X[K];/*<-*/for(int i=0;i<K;++i)X[i]=yRandom::RandU(I,0,1);
      double*r;/*<-*/NNSearchExhaustive(A,A+m,X,r,K);
      s+=*r;}
    printf("%7d   |%8.3f  |%9.6f |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock(),yRandom::Initialize(I,1);//.............begin NNSearch() test
    yKDTree::NODE<double>*R=yKDTree::NewTree(A,A+m,K);
    for(int k=0;k<M;++k){
      double X[K];/*<-*/for(int i=0;i<K;++i)X[i]=yRandom::RandU(I,0,1);
      double S=1E9;
      double*r;/*<-*/yKDTree::NNSearch(R,X,r,S,K);
      s+=*r;}
    yKDTree::DeleteTree(R);
    printf("%8.3f  |%9.6f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```



Example – Creating a Simple Voronoi Diagram

```
#include <cmath>//........................................................fabs()
#include "y_bmp.h"//..................................yBmp,<cstring>{memcpy()}
#include "y_kd_tree.h"//...............................................yKDTree
inline void Rainbow(//<========================================RAINBOW COLOR MAP
    unsigned char C[3],//<---------------------------OUTPUT COLOR (CALCULATED)
    double x,//<----------------------VALUE FOR WHICH A COLOR WILL BE CALCULATED
    double min,double max){//<-----------------MINIMUM AND MAXIMUM SCALED VALUES
  if(x<min){C[0]=C[1]=C[2]=0;/*&*/return;}//.......set too small values to black
  if(x>max){C[0]=C[1]=C[2]=255;/*&*/return;}//.....set too large values to white
  x=(1-(x-min)/(max-min))*8;//.......................remap x to a range of 8 to 0
  C[0]=int((3<x&&x<5||x>7     ?-fabs(x/2-3)+1.5:5<=x&&x<=7?1:0)*255);//.....blue
  C[1]=int((1<x&&x<3||5<x&&x<7?-fabs(x/2-2)+1.5:3<=x&&x<=5?1:0)*255);//....green
  C[2]=int((     x<1||3<x&&x<5?-fabs(x/2-1)+1.5:1<=x&&x<=3?1:0)*255);//......red
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
int main(){//<=========================CREATE A VORONOI DIAGRAM USING NNSearch()
  double*A[5],B[5][3]={4,1,0 , 4,3,1 , 6,2,2 , 2,4,3 , 8,4,4};
  for(int i=0;i<5;++i)A[i]=B[i];
  yKDTree::NODE<double>*N=yKDTree::NewTree(A,A+5,2);
  int n=1000;//................................image size will be 2n x n pixels
  unsigned char*I=yBmp::NewImage(2*n,n,255),BLACK[3]={0};
  for(int p=0;p<2*n;++p)for(int q=0;q<n;++q){
    double X[2]={p*10./(2*n-1),q*5./(n-1)};
    double S,*r;/*<-*/yKDTree::NNSearch(N,X,r,S=1E9,2);
    if(S<.002)memcpy(yBmp::GetPixel(I,p,q),BLACK,3);
    else Rainbow(yBmp::GetPixel(I,p,q),r[2],-1.5,5.5);}
  yBmp::WriteBmpFile("voronoi.bmp",I);
  yKDTree::DeleteTree(N),delete[]I;
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~15JUL2014~~~~~~
```



11. References

1. Bentley, JL. Multidimensional binary search trees used for associative searching. Communications of the ACM 1975;18:509–517.
2. Yager, RJ. Generating pseudorandom numbers from various distributions using C++. Aberdeen Proving Ground (MD): Army Research Laboratory (US); June 2014. Report No.: ARL-TN-613.
3. Yager, RJ. Reading, writing, and modifying BMP files using C++. Aberdeen Proving Ground (MD): Army Research Laboratory (US); August 2013. Report No.: ARL-TN-559.
4. Vogel, H. A better way to construct the sunflower head. Mathematical Biosciences 1979;44:179–189.

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	RDRL WML A R YAGER